Date=10/11/2020
Lecture By=Manish Mahant
Subject ⇒ Iterators and Generators

IN PREVIOUS LECTURE **(QUICK RECAP) Date-6/11/2020**	**In Today's Lecture (Overview)**
JavaScript Classes JavaScript Class Syntax Syntax Example Using a Class The Constructor Method Class Methods Syntax Example Inheritance In Javascript Prototypal inheritance Getting started Defining a Teacher() constructor function Property getters and setters Getters and setters Accessor descriptors Questions For Self-Practice// CC And ASSIGNMENT For The Day	Iterators Generator functions yield Syntax Description Questions For the Self-Practice

Processing each of the items in a collection is a very common operation. JavaScript provides a number of ways of iterating over a collection, from simple `for` loops to `map()` and `filter()`.

Iterators and Generators bring the concept of iteration directly into the core language and provide a mechanism for customizing the behavior of `for...of` loops.

For details, see also:

- `Iteration_protocols`
- `for...of`
- `function*` and `Generator`
- `yield` and `yield*`

Iterators

In JavaScript an **iterator** is an object which defines a sequence and potentially a return value upon its termination.

Specifically, an iterator is any object which implements the Iterator protocol by having a `next()` method that returns an object with two properties:

`value`

The next value in the iteration sequence.

`done`

This is `true` if the last value in the sequence has already been consumed. If `value` is present alongside `done`, it is the iterator's return value.

Once created, an iterator object can be iterated explicitly by repeatedly calling `next()`. Iterating over an iterator is said to consume the iterator, because it is generally only possible to do once. After a terminating value has been yielded additional calls to `next()` should simply continue to return `{done: true}`.

The most common iterator in JavaScript is the Array iterator, which simply returns each value in the associated array in sequence.

While it is easy to imagine that all iterators could be expressed as arrays, this is not true. Arrays must be allocated in their entirety, but iterators are consumed only as necessary. Because of this, iterators can express sequences of unlimited size, such as the range of integers between `0` and `Infinity`.

Here is an example which can do just that. It allows creation of a simple range iterator which defines a sequence of integers from `start` (inclusive) to `end` (exclusive) spaced `step` apart. Its final return value is the size of the sequence it created, tracked by the variable `iterationCount`.

```
function makeRangeIterator(start = 0, end = Infinity, step = 1) {
    let nextIndex = start;
    let iterationCount = 0;

    const rangeIterator = {
       next: function() {
           let result;
           if (nextIndex < end) {
               result = { value: nextIndex, done: false }
               nextIndex += step;
               iterationCount++;
               return result;
           }
           return { value: iterationCount, done: true }
       }
    };
    return rangeIterator;
}
```

Using the iterator then looks like this:

```
const it = makeRangeIterator(1, 10, 2);

let result = it.next();
while (!result.done) {
 console.log(result.value); // 1 3 5 7 9
 result = it.next();
}

console.log("Iterated over sequence of size: ", result.value); // [5
numbers returned, that took interval in between: 0 to 10]
    ret
```

Generator functions

While custom iterators are a useful tool, their creation requires careful programming due to the need to explicitly maintain their internal state. **Generator functions** provide a

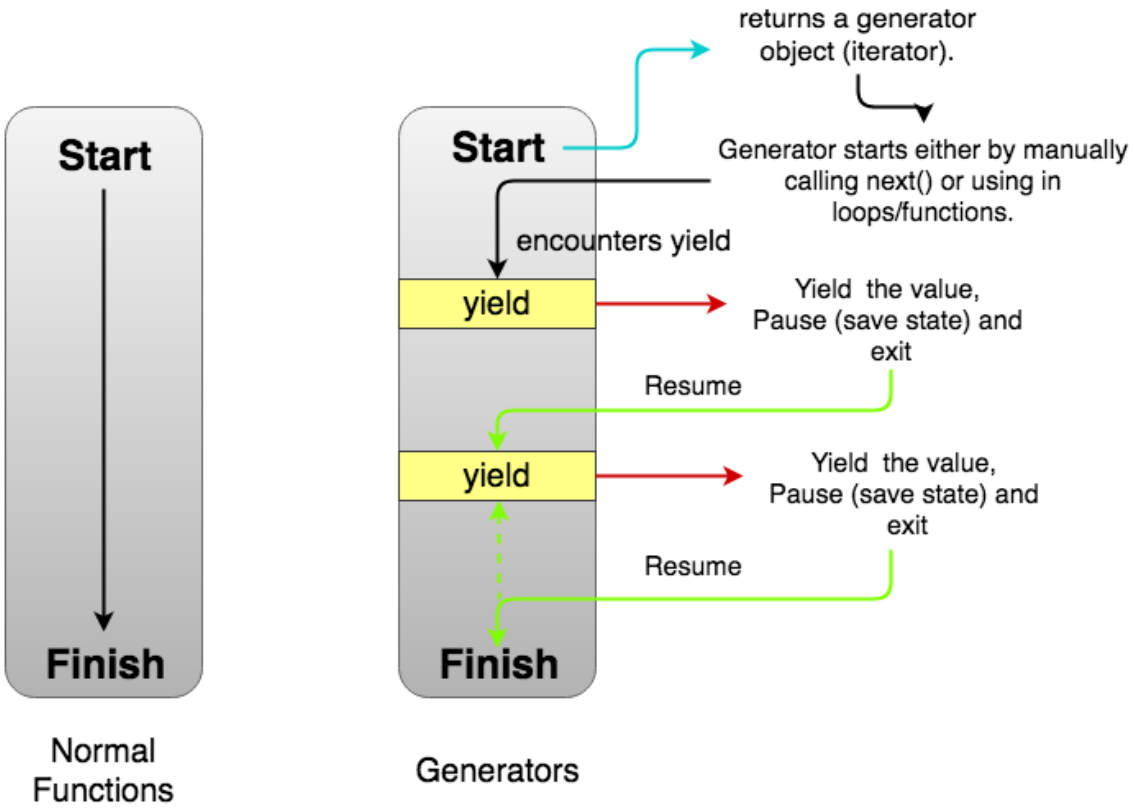
powerful alternative: they allow you to define an iterative algorithm by writing a single function whose execution is not continuous. Generator functions are written using the `function*` syntax.

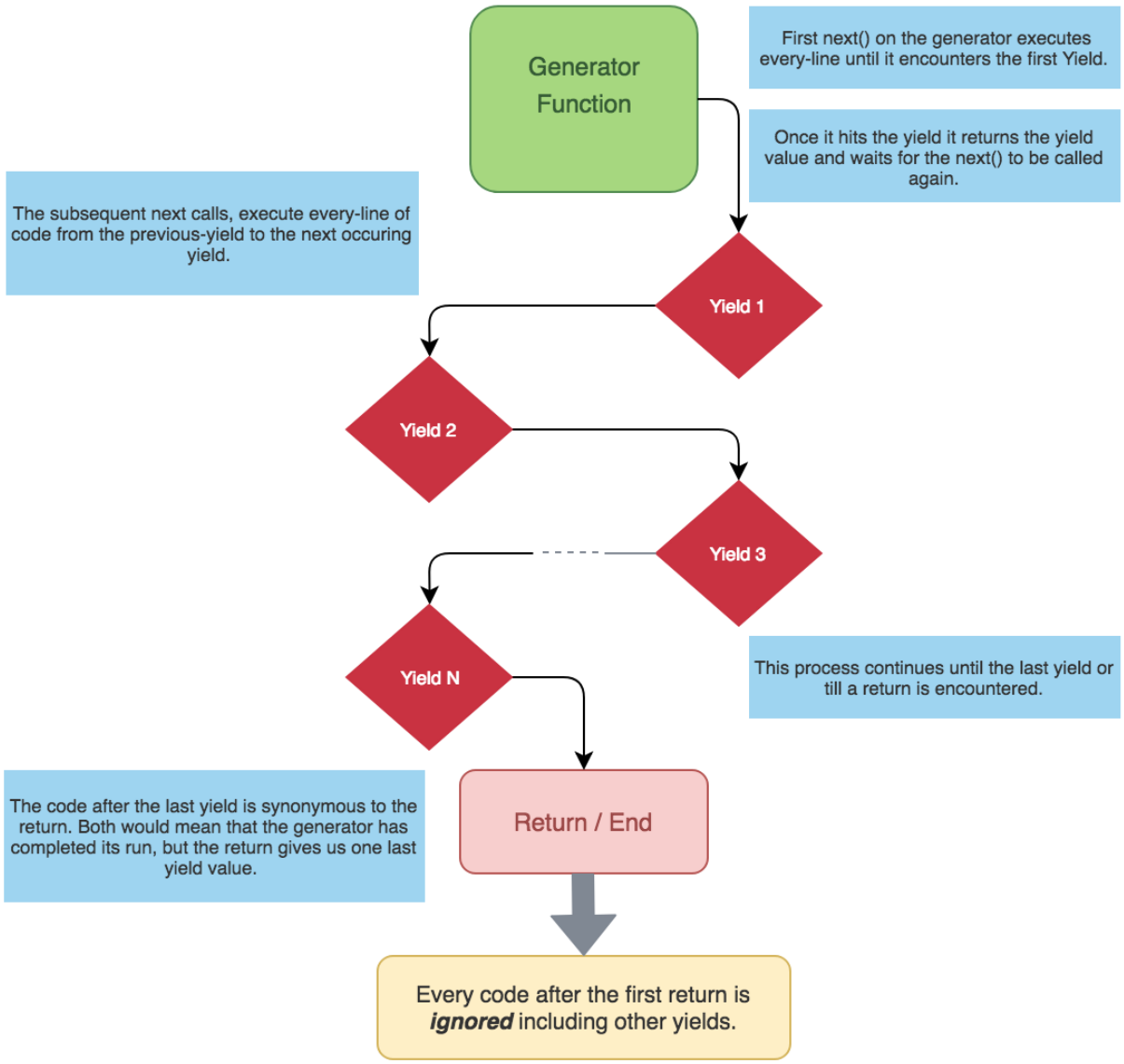
When called, generator functions do not initially execute their code. Instead, they return a special type of iterator, called a **Generator**. When a value is consumed by calling the generator's `next` method, the Generator function executes until it encounters the `yield` keyword.

The function can be called as many times as desired, and returns a new Generator each time. Each Generator may only be iterated once.

We can now adapt the example from above. The behavior of this code is identical, but the implementation is much easier to write and read.

```
function* makeRangeIterator(start = 0, end = 100, step = 1) {
    let iterationCount = 0;
    for (let i = start; i < end; i += step) {
        iterationCount++;
        yield i;
    }
    return iterationCount;
}
```


Start
Finish
Normal Functions
Start
returns a generator object (iterator).
Generator starts either by manually calling next() or using in loops/functions.
encounters yield
yield
Yield the value, Pause (save state) and exit
Resume
yield
Yield the value, Pause (save state) and exit
Resume
Finish
Generators



yield

Syntax

```
[rv] = yield [expression]
```

`expression` Optional

Defines the value to return from the generator function via the iterator protocol. If omitted, `undefined` is returned instead.

`rv` Optional

Retrieves the optional value passed to the generator's `next()` method to resume its execution.

Description

The `yield` keyword pauses generator function execution and the value of the expression following the `yield` keyword is returned to the generator's caller. It can be thought of as a generator-based version of the `return` keyword.

`yield` can only be called directly from the generator function that contains it. It cannot be called from nested functions or from callbacks.

The `yield` keyword causes the call to the generator's `next()` method to return an `IteratorResult` object with two properties: `value` and `done`. The `value` property is the result of evaluating the `yield` expression, and `done` is `false`, indicating that the generator function has not fully completed.

Once paused on a `yield` expression, the generator's code execution remains paused until the generator's `next()` method is called. Each time the generator's `next()` method is called, the generator resumes execution, and runs until it reaches one of the following:

- A `yield`, which causes the generator to once again pause and return the generator's new value. The next time `next()` is called, execution resumes with the statement immediately after the `yield`.

- `throw` is used to throw an exception from the generator. This halts execution of the generator entirely, and execution resumes in the caller (as is normally the case when an exception is thrown).
- The end of the generator function is reached. In this case, execution of the generator ends and an `IteratorResult` is returned to the caller in which the `value` is `undefined` and `done` is `true`.
- A `return` statement is reached. In this case, execution of the generator ends and an `IteratorResult` is returned to the caller in which the `value` is the value specified by the `return` statement and `done` is `true`.

If an optional value is passed to the generator's `next()` method, that value becomes the value returned by the generator's current `yield` operation.

Between the generator's code path, its `yield` operators, and the ability to specify a new starting value by passing it to `Generator.prototype.next()`, generators offer enormous power and control.

```
function* countAppleSales () {
  let saleList = [3, 7, 5]
  for (let i = 0; i < saleList.length; i++) {
    yield saleList[i]
  }
}
```

Questions For the Self-Practice

https://au-assignment.s3.amazonaws.com/Week_20_Day_2_Challenge-cd5b9f5c-26c4-4144-a8db-692c68d53bd0.pdf

https://au-assignment.s3.ap-south-1.amazonaws.com/Week_20_Day_2_Assignment-54afaeab-dc8c-4b40-ad5a-86eeaa03dc8d.pdf